68000 TRICKS AND TRAPS

BY MIKE MORTON

Some assembly language programming guidelines

THE ERA OF HIGH-LEVEL LANGUAGES has not made assembly language coding a dead art, even on modern microprocessors designed for executing compiled high-level code. Although personal computers are approaching the power of mainframes, the way to get the most out of any processor is to know when to use assembly language. The popular Motorola MC68000 processor is a good example; it has a fairly regular instruction set and instructions to support features of such languages as Pascal and C. Yet the instruction set is not perfectly orthogonal—warts in the design and implementation make the architecture interesting for hand-coded assembly programs.

The continuing usefulness of assembly language, even on this processor, is apparent in recent industry products. The Macintosh ROM, for instance, is written entirely in assembly language, yielding considerable savings in memory and speed.

In this article I'll survey some of the subtleties of the 68000 to help you avoid its pitfalls and exploit its oddities for better speed or memory use. I assume that you have some experience using the 68000; if not, see the bibliography at the end of this article.

TRAPS FOR THE BEGINNER

Most of the 68000's "traps" have a reason behind them; unintuitive aspects of the processor may actually be more useful, easier to implement, or correct in the view of the 68000 designers.

One trap is memory alignment. Although the 68000 supports byte, word, and long-word operations, word and long-word operands must be aligned on word boundaries (even addresses). This is because memory is grouped in words (2 bytes) and accessed via a 16-bit bus. Instructions must be word-aligned also, but assemblers and linkers normally do this for you.

Another trap is stack direction. The 68000 stack "grows" toward low memory. This means that to allocate stack space you should subtract from the stack pointer: SUB #size,SP. To deallocate space (or to discard previously pushed values), add to the stack pointer. Equally confusing, when allocating local storage with the LINK instruction, you must specify a negative displacement to be added to the stack pointer.

The stack pointer (SP) must stay word-aligned. If you push or pop a byte through the SP, the processor will move a word to or from the stack, placing the relevant byte in the high-order half of that word. Only the SP behaves this way; other address registers act the way you'd expect. This may seem an anomaly in an orthogonal architecture, but the SP must stay word-aligned so that words and long words are pushed to even addresses.

Shift and rotate instructions can operate on the byte, word, or long-word part of a data register, but shifts of memory operands can be only word size. Data registers can be shifted by up to 32 bits if the shift count is specified in another register or by up to 8 bits if the shift count is a constant given in the instruction. Memory can be shifted by only 1 bit.

The syntax of two-operand instructions may be reversed from other machines you're used to. For instance, the

(continued)

Mike Morton received his B.A. *in mathematics from Dartmouth College. He is currently a senior software engineer for Lotus Development Corporation* (161 First St., Cambridge, MA 02142).

68000 instruction MOVE.W D0,D1 is equivalent to LOAD D1,D0 on some other machines; that is, the contents of the D0 register are moved into D1. On the 68000, the destination register—the one affected by the instruction—is always second. The operand order for CMP instructions is also reversed from some older machines; therefore you would read CMP D0,D1 as "compare D1 to D0." (But beware: Some assemblers reverse the order of the operands from Motorola's standard; UNIX assemblers often do this.)

The 68000 provides the comparison operations shown in table 1. This includes not only all six possible relationships between two numbers, but also whether numbers are compared as signed or unsigned quantities. (Comparing the word values 0006 and FFFE hexadecimal depends on how the numbers are interpreted. If they're signed numbers, 6 is greater than -2. But if they're addresses, they're unsigned, and 0006 is a lower address than FFFE.)

The confusing thing is that the expected unsigned equivalents of BLT and BGE are not BHS (branch on higher or same) and BLO (branch on lower). Instead, Motorola uses BCS and BCC, respectively. The processor is perfectly orthogonal, providing for all types of comparisons. But the mnemonics are asymmetrical on the unsigned side (unless you use a nonstandard assembler or define your own macros).

(The distinction between signed and unsigned comparisons comes up rarely, since they are the same unless one of the values involved has the high-order bit [the sign bit] set. However, when the distinction is significant, it can lead to trouble. An operating system's disk allocator may sort disk blocks using a BGT instruction. After some years, a site tries to configure a system with more than 2^{31} bytes of disk storage. Everything grinds to a halt because BGT compares 80000010 hexadecimal to 7FFFFFF0 hexadecimal and incorrectly finds the latter address to be greater. A BHI instruction would have compared and sorted the addresses correctly.)

A note on using the condition codes: After a TST instruction, the overflow (V) condition code is cleared. This means that after TST, BLT is equivalent to BMI, and BGE is the same as BPL. Stylistically, BMI and BPL make more sense after a TST unless the value being tested is the difference of two other values.

Table 1: *This table shows which branch instructions will result in a branch taken when testing for a given relationship of* D1 *to* D0 *after a* CMP D0,D1 *instruction.*

Relationship	Signed	Unsigned
D1 < D0	BLT	BCS (branch on Carry Set)
D1 <= D0	BLE	BLS
D1 = D0	BEQ	BEQ
D1 ≠ D0	BNE	BNE
D1 > D0	BGT	BHI
D1 >= D0	BGE	BCC (branch on Carry Clear)

TRAPS FOR EXPERTS

Some quirks of the 68000 are less intuitive and regularly catch seasoned programmers. Some of these aspects of the implementation suggest design difficulties and tradeoffs in the processor; others reflect the designers' ideas on what constitutes good programming.

Addresses and data are different. Most assemblers quietly assemble MOVE #0,An as a MOVEA (move to an address register) instruction without nagging the programmer about the distinction between MOVE and MOVEA. But the 68000 treats data and address values very differently.

Address operations (MOVEA, ADDA, etc.) are never byte-size.

Word values are sign-extended to 32 bits before being used in address operations. Thus, ADDA.W D1,A2 extends the low-order word of D1 before adding it to A2. In the 68000, there is no such thing as a 16-bit address, so a word-size value is converted to 32 bits before being used in address operations.

Address operations never set condition codes; most data operations do. This is useful in subroutines that return information in the condition codes:

```
TST.W D0           ; Set condition codes to
                   ; return to caller.
MOVE.L (SP)+,A0    ; Pop return address into A0.
ADD.W #params,SP   ; Deallocate <params> bytes
                   ; of parameters.
JMP (A0)           ; Return with condition codes
                   ; still set.
```

(Note that the MOVE and ADD are translated into MOVEA and ADDA by the assembler.) The condition codes set by the TST.W are unaffected by the remainder of the exit sequence.

Another trap concerns loop operations. A loop ending with a DBcc instruction (such as DBEQ) loops until the condition cc is true; this instruction can be thought of as "decrement and branch back if condition false." This is confusing since, if you were to write out several instructions to replace a DBEQ, they would contain a BNE to jump back to the top of the loop, not a BEQ.

If the condition being tested for is not detected (or if you're using DBRA), the loop will stop when the counter reaches -1, not 0. If you want the loop always to be executed once, you should enter it at the top with the count already decremented by 1. For example, to search for the first null (zero) byte in a table of *N* bytes pointed to by A1:

```
    MOVE.W #N-1,D0   ; Start the loop counter
                     ; one too low.
LOOP                 ; Come here to test
                     ; another byte.
    TST.B (A1)+      ; Is A1's byte zero?
                     ; (Advance after testing)
    DBEQ D0,LOOP     ; If not zero AND D0 is
                     ; still >=0, loop back.
```

(continued)

This loop will execute at most *N* times. It corresponds to Pascal's "repeat. . .until" construct. For the equivalent of "while. . .do," which doesn't necessarily enter the loop:

```
    MOVE.W #N,D0         ; Start the loop counter normally.
    BRA LOOPSTART        ; Don't fudge D0; jump to
                         ; the loop end first.
LOOP                     ; This is the loop head.
    TST.B (A1)+          ; Is A1's byte zero?
                         ; (Advance after testing)
LOOPSTART                ; Enter here to check count
                         ; before looping.
    DBEQ D0,LOOP         ; If not zero AND D0 is
                         ; still >=0, loop back.
```

If you're using DBcc, don't forget to initialize the condition codes so the DBcc doesn't fall through when you jump to it. In the code above, the MOVE.W #N, D0 "primes" for the loop.

Also remember that the data register used to control the loop is decremented as a word quantity. If it's possible to have more than 2^{16} iterations, you have to nest two DBcc loops. For example, to checksum a list of bytes whose length is specified in the long word D0:

```
        MOVEQ #0, D3       ; Initialize checksum.
        MOVE.W D0,D1       ; Low word of loop length in D1.
        MOVE.L, D0,D2      ; Get high word of loop length
        SWAP D2            ; in D2 to use for outer loop.
        BRA.S START        ; Enter at the end of the loop.
LOOP:   ADD.B (A1)+,D3     ; Add the next byte into sum.
START:  DBRA D1,LOOP       ; Inner loop: Loop on low word of
                           D0.
        DBRA D2,LOOP       ; Outer loop: Loop on high word.
```

Small adjustments to the stack pointer can be done with ADDQ (or SUBQ) #n,SP, but these instructions can change it by at most 8 bytes. The fastest way to change it by more than 8 bytes is with LEA n(SP),SP.

The 68000 does not allow you to execute a MOVE instruction with a destination relative to the program counter (PC). In the view of the 68000 designers, code should not patch itself. If you must change a table in the middle of code, you must point to it with an instruction like LEA TABLE(PC),An and then alter it through An. (Self-modifying code is especially bad for 68000 programs that may someday run on the 68020, because the 68020's instruction cache normally assumes that code is pure.)

For no apparent reason, the CLR instruction always reads from an operand before clearing it. But unlike BCLR, CLR doesn't set the condition codes. Never use CLR to write a zero to a memory-mapped device address if the device will be affected by the read. The Scc instruction and MOVEs from the status register also read before writing but are less likely to cause problems.

Don't confuse the EXG and SWAP commands. EXG exchanges the 32-bit contents of two registers. SWAP swaps the 16-bit halves of a single data register.

When indexing into an array, remember to multiply the index register by the "stride" (bytes per element) of the array. For instance, if D0 holds an index into an array of long words pointed to by A1, you must multiply the index by 4 to convert from long words to bytes:

```
MULU #4,D0              ; Turn the array index into
                        ; a byte offset.
MOVE.L 0(A1,D0.L),D1    ; Pick up the long-word
                        ; array element in D1.
```

The EOR instruction must have a data register for the source, except for the immediate form of the instruction, EORI.

CODING FOR SPEED: PRINCIPLES

The secret of efficient code on the 68000 can be described using one word: "registers." Suppose, for example, that you have two 32-bit variables. If you keep them in registers, the time to add one to the other with ADD.L D0,D1 is 8 clock periods. If they're in memory pointed to by address registers, the time to add them with MOVE.L (A0),D0 and ADD.L D0,(A1) is 32 clock periods, four times slower! The moral of the story is simple: Work hard to keep frequently used quantities in registers.

You can learn this important rule and others by studying instruction timing information (such as the tables in the *M68000 16/32-bit Microprocessor Programmer's Reference Manual*). Times are given in clock periods, which I'll call cycles; a 10-MHz processor executes 10 cycles per microsecond. In general, the tables give the base time for each instruction. Most base times must have additional times added in for the operands. For instance, the time to execute AND.W D0,(A1)+ is 8 cycles for a word-size AND-to-memory and 4 more cycles for the (A1)+ destination operand. (The source operand is "free" because it's a register.) Thus, the entire instruction takes 12 cycles, or 1.2 μsec on a 10-MHz processor.

When you're trying to save a few cycles in a crucial loop, timing tables can be useful as more than just a reference. They provide a concise summary of the architecture—sort of a shopping list of the instructions available and the cost of each. When you're trying to avoid preconceived notions of which instructions are suited to solving a problem, this summary can remind you of alternatives and encourage lateral drift in your thinking.

CODING FOR SPEED: BASIC RULES

The MOVEQ, ADDQ, and SUBQ instructions are great. For instance, it's faster to zero all 32 bits of a data register by using MOVEQ #0,Dn than it is to use CLR.L Dn. Remember that these instructions are limited to small numbers: MOVEQ can load values from −128 to 127 into a data register; ADDQ (SUBQ) can add (subtract) only values from 1 to 8 to (from) its destination.

DBcc is especially efficient; use it whenever you can. (But beware the traps described above.)

Not all assemblers automatically produce "short" branches (branches with 8-bit displacements). Check the output of your assembler to see if it emits a short branch whenever possible. If not, you may have to use BRA.S, BSR.S, and Bcc.S in your source code instead of BRA, BSR, and Bcc.

(continued)

Because a taken short branch is slower than an untaken one, try to avoid taking most branches. For instance, if you have a loop searching for a null, the simple way to search is with

```
LOOP                ; Here to search for the
                    ; next null.
    TST.B (A3)+     ; Check next byte; advance
                    ; the search pointer.
    BNE.S LOOP      ; Loop back if not found.
```

It takes only a bit more space to "unroll" one or more iterations of the loop:

```
LOOP                ; Here to search for the
                    ; next null.
    TST.B (A3)+     ; Check next byte; advance
                    ; the search pointer.
    BEQ.S FOUND     ; If zero, exit the loop.
    TST.B (A3)+     ; Not zero: check another
                    ; byte and advance.
    BNE.S LOOP      ; If still not found,
                    ; loop back.
FOUND               ; Come here when A1 points
                    ; one past the null byte.
```

If the character tested generally isn't zero, the BEQ.S usually goes untaken and is faster. You can unroll any number of iterations, adding TST.B/BEQ.S pairs until the extra space consumed is no longer worth the diminishing increase in speed (or the branches become long branches).

Addressing with (An)+ is faster than with –(An). If you have the choice of which direction to go in a search or other loop through memory, move upward. (Note that this is not true for the destination operand of a MOVE instruction.)

Because (An) addressing is faster than x(An), access to the first element of a data structure is faster than to the others. (This is also useful with Pascal records, C structures, etc.)

The MOVEM instruction is a very efficient way to stack or unstack a large number of registers. But if you have to push only two registers, or pop three, MOVEM is no faster than moving them one at a time.

Don't assume that long operations are always slower than word-size ones. For instance, word address operations can be slower than long ones because of the time to sign-extend a word value.

As with other machines, never multiply or divide by a power of 2 when you can shift instead. Although shifts are time-consuming, they're always faster than a multiplication or division. So, you can use the ASL (arithmetic shift left) instruction to multiply by a power of 2 and use ASR (arithmetic shift right) to divide by a power of 2. (Be careful here—the right shift is not the same as a division if the contents of the register are negative. For example, –1 divided by 2 should be zero, but –1 shifted right by 1 bit is –1, rounded incorrectly.) Don't forget that the multiplication instructions produce a long-word result from a word operand; shifting doesn't.

To multiply by 2, add a register to itself instead of shifting: ADD Dn,Dn. In fact, if you are multiplying a word operand by 4, you can do it faster with two ADD instructions than with a single shift by 2 bits.

Similarly, in doing extended-precision arithmetic, you can replace the common operation ROXL #1,Dn with ADDX Dn,Dn and save 2 or 4 cycles, depending on whether the operands are words or longs.

You can compute certain multiplications faster with shifts even if they're not powers of 2. For instance, to multiply D0 by 17, add D0 to 16 times D0:

```
MOVE D0,D1          ; Copy D0 to D1.
LSL #4,D0           ; Compute 16 x D0 in D0.
ADD D1,D0           ; Add original value in to
                    ; compute 17 x D0.
```

Computing products this way is still faster than the 40-plus cycles for a multiply instruction.

The cost of maintaining the stack can be lessened if arguments are deleted after the call by the caller, not the subroutine. (Most C compilers use this stack protocol.) If the stack doesn't have to be cleaned up after every call, you can allow debris from several calls in a row to accumulate as long as it's easy to keep track of how much there is. Typically, you can let it pile up until you reach a branch, then unstack it all with an ADDQ (or LEA if there's more than 8 bytes to remove).

Finally, don't ignore the 68000's "higher-level" instructions. Even at the assembler level, instructions such as PEA, LINK, UNLK, and CHK can be very useful.

Coding for Speed: Some Cookbook Examples

Here are a variety of things you can do to save time when you're scraping for cycles. Some are useful in many applications; others are very specialized. The more obscure ones are examples of the kinds of tricks that the 68000 can do.

Remember that timings will not be the same on the 68000's relatives (the 68008, 68010, 68020, etc.). If you're working on one of these processors, recompute the timings or, when you're not sure which of two approaches is faster, measure the speed of both. Timings for the 68020 will be especially hard to compute because of its sophisticated prefetch and instruction caching.

You should also know that not all computers run the processor at the advertised speed. For instance, the Macintosh's 68000 runs at 7.8 MHz, but it can't always operate at this speed because the screen is memory-mapped and "steals" some memory cycles. Thus, the effective speed of the Macintosh is about 6 MHz, but only memory cycles are slowed down—CPU cycles are unaffected. So operations done mostly within the CPU (such as multiply, divide, and long register shifts) run at nearly full speed. The lesson in all this is that timings are hard to compute or intuit; you may want to time various pieces of code for yourself to see which is faster.

It is said that one doesn't really know how to use a tool until one knows three ways to abuse it. Here are some of my favorite ways to abuse the 68000.

(continued)

Fast subroutine calls. Although JSR and RTS provide a simple subroutine call-and-return, the cost of pushing the return address on the stack is significant. For a very frequently called subroutine, you can change the call to store the return in an address register as in

```
  LEA RETURN,A0     ; Return address goes in A0.
  JMP routine       ; Jump to the subroutine.
RETURN              ; A0 points to this spot.
```

Then to return, just JMP (A0). By avoiding use of memory, this saves 8 cycles. (Note that the LEA instruction references the label RETURN with PC-relative addressing.)

Also, if you end a subroutine with

```
JSR lastsub         ; Call one last subroutine
RTS                 ; and return.
```

and lastsub doesn't alter the stack, you can save a whopping 24 cycles by using "tail recursion" to replace the two instructions with a single

```
JMP lastsub         ; "Call" lastsub and
                    ; it'll return for us.
```

Finally, if you call a subroutine and then branch somewhere else, you can avoid extra jumping around. For instance,

```
JSR sub             ; Make a call
JMP next            ; and go somewhere else.
```

can be made slightly faster with

```
PEA next            ; Push a fake return address
JMP sub             ; and "call."
                    ; sub will RTS to next for us.
```

(All of the above work for BSR and BRA as well as JSR and JMP.)

Quick test for zero. If you want to test whether a register is zero and don't mind trashing the value, use DBRA Dn,NOTZERO instead of combining TST.W Dn with BNE NOTZERO.

If you want to do an *N*-way branch depending on a value, you'll usually want to index into a jump table and transfer to the appropriate address. A "case" statement is typically implemented this way. But if you have a very small number of values and want to handle the lower values more quickly, a series of DBRAs can do this conveniently. For example, if you want to branch based on a register that contains 0, 1, or 2,

```
  DBRA D0,NOT0      ;Decrement; jump if it wasn't
                    ; zero.
    <handle zero case>
NOT0                ; Come here if not zero.
                    ; D0 has been decremented.
  DBRA D0,NOT1      ; Decrement; jump if original
                    ; D0 wasn't one.
    <handle one case>
NOT1                ; Not one. D0 has been
                    ; decremented twice.
  DBRA D0,ERROR     ; Decrement; if not originally
                    ; two, error.
    <handle two case>
```

Checking for membership in a small set. If you want to see if a number is in a set of several numbers, you can create a bit mask corresponding to the set. For instance, if the set is {0,1,3,5}, the mask has those bits set and the bit map is 00101011 (2B hexadecimal). You can test for membership in this set with

```
BTST D0,#$2B        ; Is D0 in {0,1,3,5}?
```

If your set is composed of more than eight elements you have to move the mask into a data register first.

Quick comparisons. To check the value of a data register with CMP.L #xxx,Dn takes 14 cycles. If the value be-

(continued)

GREATER PERFORMANCE...AT HALF THE COST!

DATA I/O 29B™ COMPATIBILITY-UNDER $1,500

Programming A World of Technologies

UNIVERSAL, GANG & SET PROGRAMMING FOR LAB AND LIGHT PRODUCTION

BYTEK ALWAYS SETTING HIGHER STANDARDS

- Supports MOS/CMOS EPROM/EEPROMs including Megabit devices. Options: Bipolar PROMs, Logic Array Devices, 40 pin Micro Chips, and GangCel
- Memory capacity to 1Mbyte, 64 bit wide programming, device type and parameters stored in non-volatile memory
- 3-Way device type selection: manually, auto ID or from menu
- 2-Way command execution: Function keys or menu
- Data I/O PROMlink™ and BYTEK PROMsoft™ compatible

25 Key-32 Character LCD display - Memory expandable to 1 Megabyte
RS232 PC Interface, Parallel Printer Port

Data I/O, PROMlink are trademarks of Data I/O, Inc.
MADE IN THE U.S.A.

21 Day No Obligation Trial Period

$1,495
COMPLETE
Model:
S125-GL

ORDER TODAY
800-523-1565 / In Fla. — 1-305-994-3520
Telex: 4998369 BYTEK

Mastercard or Visa Welcome

BYTEK Instrument Systems Division • 1021 South Rogers Circle • Boca Raton, Florida 33431 ing tested for is small enough to fit in a MOVEQ, it's shorter and faster to put the value in a temporary register:

```
MOVEQ #xxx,D0      ; Set up value to look for
CMP.L D0,Dn        ; and do the comparison.
```

If the value xxx is between −8 and 8, and you don't mind altering the data register, you can just use SUBQ #xxx,Dn (or ADDQ, as appropriate) instead of a CMP. Then you can use a conditional branch just as you would after a CMP. This works for word or long-word comparisons.

Picking up an unaligned word. The straightforward approach is to load 1 byte, shift it into position, and load the second byte. The faster way (28 cycles instead of 38) is to exploit the stack pointer's odd behavior when byte quantities are pushed on the stack:

```
MOVE.B (A0)+,-(SP) ; First byte to high half of
                   ; new word on stack.
MOVE.W (SP)+,D0    ; Pop that new word to D0.
                   ; First byte in place.
MOVE.B (A0),D0     ; Second byte in place.
```

Clearing address registers. MOVE.L #0,An takes 12 cycles, while SUB.L An,An takes only 8 and is shorter. (CLR doesn't work with address registers.)

Avoiding long shifts and rotates. The time the 68000 takes to shift a register is proportional to the distance being shifted: 2 additional cycles for every bit. Thus, never rotate a long word more than 16 bits in either direction or a word more than 8 bits. (Remember that to shift by more than 8 bits, you have to put the shift count into a data register. In the examples that follow, the bit count is *not* a constant; the value is bracketed to show this.)

```
ROL.L <16+x>,Dn = ROR.L <16-x>,Dn
ROL.W <8+x>,Dn  = ROR.W <8-x>,Dn
```

In shifting 16 bits or more, the first 16 bits of the shift can be done with a SWAP to save 26 cycles in each of these cases:

```
LSR.L <16+x>,Dn =
  CLR.W Dn          ; Clear bits that swap up
  SWAP Dn           ; and LSR.L #16,Dn.
  LSR.W <x>,Dn      ; Now finish the shift.

ASR.L <16+x>,Dn =
  SWAP Dn           ; Slide down 16 bits.
  EXT.L Dn          ; Sign-extend to a long word.
  ASR.W <x>, Dn     ; Finish up, sign-extended.

LSL.L <16+x>,Dn =
  LSL.W <x>Dn       ; Shift x bits in low half.
  SWAP Dn           ; Shift 16 more bits.
  CLR.W Dn          ; Throw away bottom half.
```

And some long-word operations of less than 16 bits can be optimized with SWAP. Long shifts between 11 and 15 bits can be speeded up with

```
LSL.L <x>,Dn =
  SWAP Dn              ; Rotate left by 16 bits.
  ROR.L <16-x>,Dn      ; Undo to x-bit left rotate.
  AND.W #mask,Dn       ; Remove bottom x bits.

LSR.L <x>,Dn =
  AND.W #mask,Dn       ; Remove bottom x bits.
  SWAP Dn              ; Rotate right by 16 bits.
  ROL.L <16-x>,Dn      ; Undo to x-bit right rotate.
```

Fast sign-extend. While there are instructions to sign-extend bytes into words or words into long words, what if you have a signed 12-bit field (from unpacking a record or reading a DAC)? The standard way to sign-extend this to a full 16-bit word is with

```
LSL.W #16-12,Dn  ; Shift so 12-bit field is left-justified.
ASR.W #16-12,Dn  ; Shift it back down sign-extended.
```

If you know that the bits outside the 12-bit field are zero, you can do this without shifting. In general, if you want to sign-extend an *N*-bit field to 16 bits, define "mask" to be $-(2^{(N-1)})$—a mask with the bottom *N*+1 bits clear. Then the sign extension can be done using a temporary register:

```
MOVE.W #mask,D0   ; Build mask with high N+1 bits set.
ADD.W D0,Dn       ; Negative: top bits=0.
                  ; Positive: top bits=1.
EOR.W D0,Dn       ; Flip so top bits are correct.
```

This is always at least as fast as the shifting method, which gets slower as *N* increases. Sign-extending to a long word is faster this way if *N* is 3 or more.

Loading large constants. To move certain values into the upper half of a data register, you might code MOVE.L #00xx0000,Dn. It's faster to replace this single instruction with two:

```
MOVEQ #xx,Dn    ; Move value to lower half
                ; and clear upper half.
SWAP Dn         ; Swap—put things in position.
```

Clearing the upper half of a data register. Instead of doing this with AND.L #$FFFF,Dn, it's quicker to use

```
SWAP Dn    ; Swap high and low halves.
CLR.W Dn   ; Clear high half while it's low.
SWAP Dn    ; Put things back in place.
```

CONCLUSIONS

Esoteric coding techniques continue to be important in pushing processors to their limits. A machine such as 68000, which is oriented toward executing compiled high-level languages, can still be appropriate for tight hand-crafted solutions. A programmer who needs the utmost in performance can exploit quirks in an instruction set to great advantage. ■

BIBLIOGRAPHY

Harmon, Thomas L., and Barbara Lawson. *The Motorola 68000 Microprocessor Family*. Englewood Cliffs, NJ: Prentice-Hall, 1985.

Kane, Gerry, Doug Hawkins, and Lance Leventhal. *68000 Assembly Language Programming*. Berkeley, CA: Osborne/McGraw-Hill, 1981.

Motorola Inc. *M68000 16/32-bit Microprocessor Programmer's Reference Manual*. Englewood Cliffs, NJ: Prentice-Hall, 1984.

Scanlon, Leo J. *The 68000: Principles and Programming*. Indianapolis, IN: Howard W. Sams, 1981.

Starnes, Thomas W. "Design Philosophy Behind Motorola's MC68000." BYTE, April–June 1983.

Williams, Steve. *Programming the 68000*. Berkeley, CA: Sybex, 1985.